MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1020 February 1988

System Validation via Constraint Modeling

by

Richard C. Waters

Abstract

Currently, there are two major approaches to system validation: testing and code inspection. Each of these methods indirectly checks the correctness of a system by attempting to find faults in the system. It is advantageous to apply both methods to a system because the strengths and weaknesses of the methods are complementary. Testing is good at finding failures in the usual operation of the system even if these failures are created by complex interactions between modules. Code inspection is good at finding local faults in single modules even if these faults only manifest themselves as failures in unusual situations.

Constraint modeling could be an important third method of system validation. The essence of constraint modeling is the creation of a model that represents key aspects of the behavior of a system, while ignoring other aspects. Given the model, constraint propagation can be used to detect inconsistencies in the operation of the system. The advantage of constraint modeling as a means of system validation is that it is complementary to both testing and code inspection. In particular, constraint modeling can locate errors even if they are caused by non-local faults and manifest themselves as failures only in unusual situations.

As a result, even though the ability of constraint modeling to find errors is limited both by the simplifications which are introduced when making the model and by the power of the constraint propagator available, constraint modeling has the potential for significantly increasing overall system reliability when used in conjunction with testing and code inspection.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the National Science Foundation, of the IBM Corporation, of the NYNEX Corporation, of the Siemens Corporation or of the Department of Defense.

Current Approaches to System Validation

As used here, the term *system validation* denotes the process of checking that a system satisfies its specifications. In contrast to *verification* (which denotes proving that a system satisfies its specifications) validation methods take the basic approach of looking for potential failures in a system. If no failures are found, then a certain degree of confidence in the validity of the system is justified. However, using this kind of approach, it is never possible to obtain complete confidence in a system, because it is never possible to look for every conceivable failure.

Testing. The most common form of system validation is testing. Given the specification for a system, test cases are created which embody microscopic facets of the specification. Each test case specifies a situation (e.g., a set of input data) and what is supposed to happen in that situation (e.g., a set of output data). In order to use a test case to check a system, one need only put the system in the specified situation (e.g., by feeding it the appropriate input data) and observe the results.

If the system fails a test, then one can say with total confidence that a failure has been detected. However, it is more difficult to evaluate exactly what it means when the system passes a test. At a minimum, this means that the system is correct with regard to the particular specification facet captured by the test case. However, this is of limited value, because the specification facets captured by individual test cases are extremely small.

Fortunately, one can take advantage of the fact that programs are basically quite orderly in nature to conclude that if a system works correctly in a particular situation, it will most likely work correctly in *similar* situations. (This statement is very vague, but captures the spirit of an important point. Much of the research on test case generation can be viewed as investigating exactly what it means for two situations to be similar. At the least, two situations cannot be similar unless they lead to the same execution path through a program and trigger the same implications in the specification.)

For testing to be effective, one must create a suite of test cases that covers a wide range of situations. That is to say, a significant percentage of all possible situations must be similar to situations which are actually tested by the suite. In order to satisfy this criterion, one must have a good understanding of what situations are possible and which situations are similar. Unfortunately, there is no short cut to obtaining such an understanding. Although there are a tools which can be of assistance (e.g., tools which ensure that a suite of tests cases exercises every statement in a system), creating a good suite of test cases requires an in depth understanding of both the system and its specification, and is more of an art than a science.

Given a sufficiently large set of well-designed test cases, testing can be very effective as a method for validating the operation of a system in usual situations. However, there are a number of reasons why testing is relatively ineffective as a method for validating the operation of a system in unusual situations.

Suppose that there is a system which exhibits a failure f in a situation s. This failure will not be detected by testing unless the suite of test cases contains a test corresponding to some situation s' which is similar to s. Suppose that s and the situations similar to it are unusual, in that they are very unlikely to occur during the usual operation of the system. If this is the case, the ability of testing to detect f depends upon the ability of the test case writer to imagine that situations like s are possible. Perhaps more importantly, the ability

of testing to find f also depends on the extent to which the tester is motivated to seek out unusual situations to test.

Beyond the basic limits imposed by the tester's imagination and motivation, there are a number of subsidiary reasons why unusual situations tend to be inadequately tested. In particular, there are almost always many more unusual situations than usual ones. Given the limited amount of resources typically available, it is often not possible to write a full range of tests corresponding to unusual situations, nor would it be possible to run such a large suite of test cases. In addition, unusual situations often correspond to highly elaborate input scenarios. This makes it particularly hard to devise tests for these situations and particularly expensive to run the tests which are devised.

The fundamental law of faults. It is useful to make a distinction between faults, errors, and failures (see [1]). A *failure* is an externally visible incorrect behavior of a system. An *error* is an incorrect internal state which may or may not be externally detectable as a failure. A *fault* is a mistake in a program which causes one or more errors and failures.

On the surface, the goal of testing is the detection of failures. However, its real goal is the detection faults. This is so, because the only way to correct failures is to correct the underlying faults, and if all the faults are fixed, all the failures will be fixed, whether or not they have been detected. Fortuitously, testing is actually better at finding faults then failures. This is so, because of what could be termed *the fundamental law of faults*:

Most faults cause a lot of errors and failures.

At first glance, it might appear disadvantageous that single faults cause lots of failures. However, it makes faults much easier to detect. In particular, testing can determine that there is a fault as long as it can detect any of the failures it causes. Put another way, a fault is hard to detect only if *every* failure it causes is hard to detect. There are, of course, all too many faults of this kind. However, things would be much worse if failures were truly independent of each other.

Code inspection. A form of system validation which complements testing is code inspection. In this process, one or more people study a system and its specification and attempt to satisfy themselves that the system is correct. This inspection usually combines a search for errors using simulated step by step evaluation with a direct search for specific kinds of common faults. This process is most rigorous when it is applied by people other than the programmers who wrote the system. However, code inspection is also vitally important as the primary means by which programmers develop confidence in the code they write.

When code inspection identifies a fault, one can say with total confidence that this fault exists. However, as with testing, it is not easy to evaluate exactly what it means when a code inspection fails to find any faults. Unfortunately, it seldom means that there are no faults. More commonly, it merely means that there are no faults which are *easy to find.* This is a step in the right direction, but far from fully satisfactory.

The nature of human information processing abilities places significant limits on the usefulness of code inspection. In particular, people are quite limited in the total amount of detail they can attend to at a given moment. As a result, code inspection is limited to comparing individual subroutines (or small modules containing a few subroutines) with their specifications. Given a sufficiently small context, code inspection is very effective at finding faults. However, code inspection is relatively ineffective at finding errors which stem from

the interaction between widely separated subroutines. The fundamental problem is that it is all too likely that the individual subroutines are reasonable enough by themselves and that no inspector will ever look at the subroutines simultaneously.

In addition to being limited to relatively local inspections, people are limited in the complexity of the analyses they can perform. If a fault is abstruse enough, it will be missed even if it is entirely local. Further, code inspection is inherently rather informal. There is nothing beyond discussion between the members of the inspection team to ensure that the inspectors have considered all of the relevant aspects of a subroutine.

The advantages of a combined approach to validation. The usefulness of testing as a method for system validation is limited by the fact that, after a certain point, the benefit derived from additional amounts of testing diminishes rapidly. A reasonably good set of test cases can find a reasonably large number of faults reasonably cheaply. However, attempting to go very far beyond this rapidly gets to the point where very large amounts of effort are required to find each additional fault.

The diminishing returns from testing stem from the fact that, as discussed above, testing is good at finding some kinds of faults and bad at finding other kinds. The same can be said of code inspection. As a result, code inspection suffers from diminishing returns as well.

Although testing and code inspection both suffer from diminishing returns, these two approaches are largely orthogonal in that the degree of effort needed to find a given fault with testing is largely unrelated to the degree of effort needed to find the same fault using code inspection. In particular, faults which are hard to find with testing (e.g., faults linked to unusual situations) are sometimes much easier to find with code inspection and faults which are hard to find with code inspection (e.g., non-local faults) are sometimes much easier to find with testing.

The orthogonality of testing and code inspection suggests that it is much better to apply a given amount of system validation resources half to testing and half to code inspection, rather than exclusively to either approach alone. Such a combined strategy can significantly increased confidence in a system. One can be reasonably confident that every fault that is easy to detect with either method has been found.

This paper proposes a third approach to system validation (constraint modeling) which is orthogonal to both testing and code inspection. In particular, constraint modeling provides an avenue for detecting the kinds of faults which tend to elude both testing and code inspection (e.g., non-local faults linked to unusual situations). Constraint modeling holds the promise of further increasing the level of confidence obtainable from a given amount of effort by dividing this effort among three separate approaches.

Constraint Modeling

Constraint modeling is a well-known technique in artificial intelligence for performing partial reasoning about complex artifacts [3, 10, 11, 12]. The following section provides a tutorial description of constraint modeling. Readers who are familiar with this technique can skip directly to the next section.

There are two basic parts to constraint modeling. The first consists of creating a model which represents some of the key features of an artifact while ignoring others. The second (constraint propagation) is a particular kind of efficient reasoning method which can be used to draw conclusions about the model.

As an example of constraint modeling, consider the simple electronic circuit fragment in Figure 1. At the top of the circuit, there is a power bus p, which is maintained at 5 volts. At the bottom, there is a ground bus g, which is maintained at 0 volts. A 1000 ohm resistor r and a transistor t are connected in series between power and ground. A wire labeled x comes in from the left side of the figure and is connected to the base of t. A wire labeled y is connected to the junction j which connects r and t. The other end of this wire is connected to the base of a second transistor u.

Even given a circuit fragment as simple as the one in Figure 1, it is quite difficult to analyze the behavior of the circuit in full detail. However, constraint modeling can be used to draw conclusions about key aspects of this behavior.

Constraint models. Figure 2 shows an example of a constraint model (or network) corresponding to the circuit in Figure 1. Each component in the circuit is represented by a node in the model (represented as a box in the figure). These components consist of the power bus p, the ground line g, the junction j, the resistor r, and the transistors t and u. The terminals of each device are given identifying names. For example, the transistor t has three terminals: the base b, the emitter e, and the collector c. Each terminal is represented in the model by two ports: one corresponding to the voltage V and one corresponding to the current I. For example, the port V_t^b represents the voltage at the base of the transistor t. The wires in the circuit are represented by connecting links in the model which indicate equality between ports. The values V_x, I_x, V_y, and I_y are identified as being of particular interest.

To the right of each node is the set of constraints associated with the node. These constraints (or transfer functions) describe the behavior of the component being modeled. An item in the circuit is modeled using a node with constraints (as opposed to links between ports of nodes) if and only if the constraints which are required to describe the item are more complex than mere equalities. This is the reason why the junction j is modeled as a node while the wires in the circuit are not.

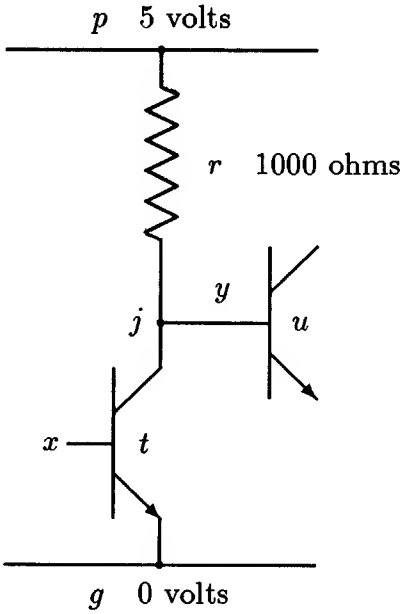

Figure 1: A simple circuit.

Consider the various sets of constraints in turn. The power bus is modeled as maintaining a voltage of 5 volts no matter what the current is—i.e., $V_p = 5$ and I_p is unconstrained. Similarly, the ground line is modeled as maintaining a voltage of 0 no matter what the current is.

The constraints on r describe the basic characteristics of a resistor. The first constraint ($I_r^a = I_r^b$) specifies that the current flowing into the resistor is always the same as the current flowing out. (In the model, current is considered to be positive when it flows from upper left to lower right.) If I_r^a and I_r^b are both known, then this constraint can be used as a consistency check. If $I_r^a \neq I_r^b$ then something is seriously wrong. If either I_r^a or I_r^b is known, then the constraint can be used to calculate the other quantity. As a result, the constraint embodies two different transfer functions. If neither I_r^a nor I_r^b is known, the constraint has no immediate implications.

The second constraint on r ($I_r^a < .01$) is an applicability condition. If the current is greater than .01 amps, the resistor will overheat. The third constraint ($V_r^a - V_r^b = I_r^a * 1000$) relates the current through the resistor and the voltage drop across the resistor. The constraint implies three different transfer functions. If any two of the quantities V_r^a, V_r^b, and I_r^a are

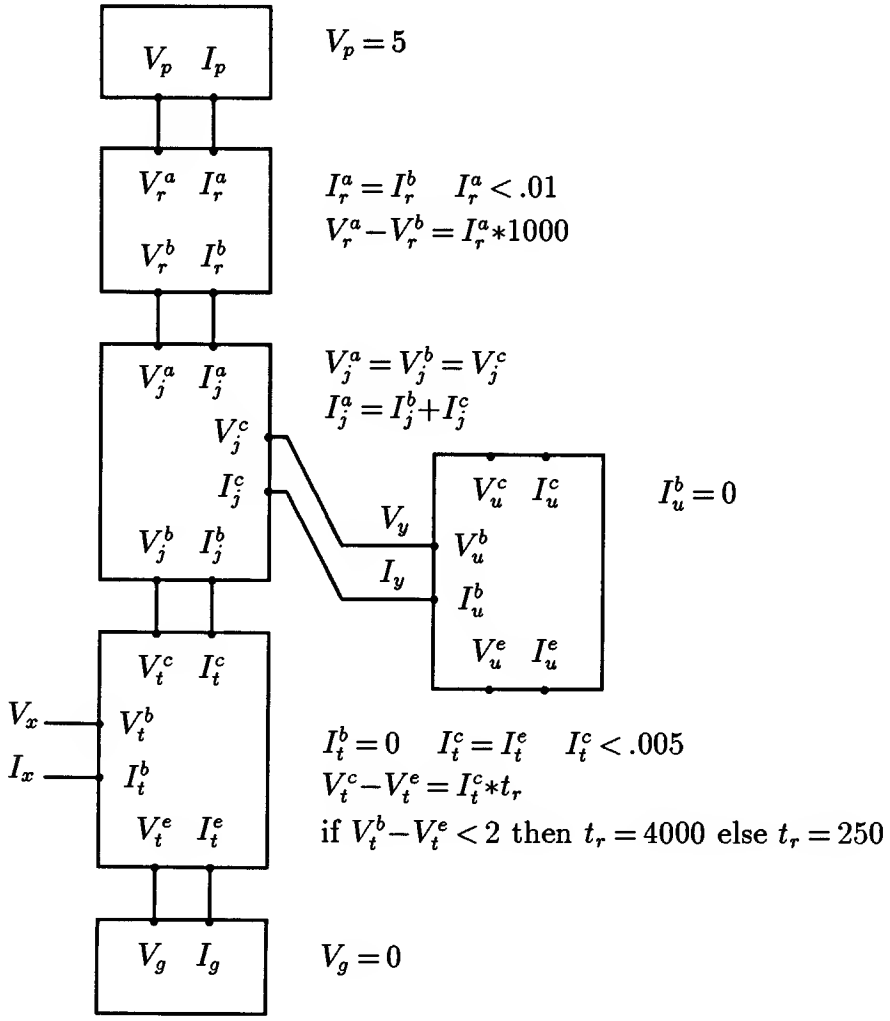

Figure 2: A constraint model of the circuit in Figure 1.

known, then the third one can be calculated.

$$\begin{aligned} V_r^a &= I_r^a * 1000 + V_r^b \\ V_r^b &= V_r^a - I_r^a * 1000 \\ I_r^a &= (V_r^a - V_r^b)/1000 \end{aligned}$$

The constraints on the junction j specify that the voltage on all the wires at the junction must be the same and the current entering the junction must be equal to the current leaving the junction. As above, the second constraint implies three transfer functions. If any two currents are known, the third can be calculated.

The constraints on the transistor t specify a simplified model of its behavior. The first two constraints state that the current through the base is always zero and that the current entering the collector is the same as the current leaving the emitter. The third constraint ($I_t^c < .005$) is an applicability condition which reflects the fact that the transistor will overheat if too much current is applied.

The last two constraints represent the transistor as a variable resistor with two states. If the voltage between the base and emitter is low, then the resistance between the collector and emitter is high. Conversely, if this voltage difference is high, the resistance is low. Together, these constraints imply four transfer functions. For example if V_t^b, V_t^c, and V_t^e are known, I_t^c can be calculated.

A particularly interesting situation arises if V_t^c, V_t^e, and I_t^c are known. In this case, the constraint can be used both as a consistency check and as a transfer function. The values of V_t^c, V_t^e, and I_t^c must be consistent with one of the two states of the transistor. If they are, then it is possible to determine whether or not $V_t^b - V_t^e < 2$. (Transfer functions yielding this kind of partial information are actually more common than transfer functions yielding completely precise results.)

The constraints on u are analogous to the constraints on t. (In general, the form of the constraints is the same for every node of a given type.) However, only one of these constraints is relevant to the examples below. In the interest of simplicity, only this one constraint is shown in the figure.

Constraint propagation. Given the constraint model in Figure 2, the behavior of the circuit in Figure 1 can be analyzed using constraint propagation. Constraint propagation consists of two kinds of steps: information transmission (over links between nodes) and information utilization (using the constraints at individual nodes). Constraint propagation efficiently computes global conclusions by combining local deductions performed at individual nodes. The efficiency of the algorithm stems from the fact that each utilization step is small and local and the topology of the model limits the effects which a transmission step can have.

Information transmission steps are trivial operations. If a piece of information exists at one end of a link and not at the other, then it is transferred across the link. (This can be implemented very efficiently by using a single memory cell which is shared by all of the ports connected to the link.) Once the piece of information reaches the other end of the link, it may or may not be utilized by the destination node. If it is, then new information may appear at one or more ports of the destination node. This information may trigger additional transmission steps which may trigger additional utilization and so on. Cycles of constraint propagation continue until the model reaches a state where no new information is created at any node.

For example, in Figure 2, constraint propagation would begin with the following five steps of information transmission: $I_t^b = 0 \rightarrow I_x = 0$, $I_u^b = 0 \rightarrow I_y = 0 \rightarrow I_j^c = 0$, $V_p = 5 \rightarrow V_r^a = 5$, and $V_g = 0 \rightarrow V_t^e = 0$.

Information utilization. The question of what happens once a new piece of information reaches a node is inherently much more complex. There are two basic ways in which information is utilized: to compute additional information and to detect contradictions.

The preceding discussion of Figure 2 presupposes a particularly simple kind of utilization. It is assumed that all of the information which is transferred between nodes is in the form of concrete numbers and that these numbers are utilized by detecting inequalities between numbers and by computing new numbers. There are three possibilities which have to be considered whenever a number is transmitted over a link:

(A) The value may already have been stored at the end of the link. In this situation, no action need be taken.

(B) There is already a value at the end of the link, but this value is different from the transmitted value. In this situation, a contradiction has been detected—there is something wrong with some assumption which has been made.

(C) There was no previous value at the end of the link. In this situation, the constraints on the node have to be checked to see if they are consistent with the new datum. If any constraint is violated, then a contradiction has been detected. In addition, the constraints on the node have to be checked to see if any of them can be used to compute new information at some port of the destination node. If this is possible, then these new values are recorded and propagation must continue starting with these new values.

Examples 1, 2, and 4 below show how simple numerical propagation operates. In these examples it should be noted that the order of propagation has been chosen to facilitate the exposition. In actual fact the propagation would occur in a less ordered way. However, the propagation process is still very efficient.

Complex utilization. Constraint propagation of numerical values (or other literal quantities) is efficient and can be useful in many situations, however, it is limited in the power of the deductions which can be performed. There are a wide variety of ways in which more complex information can be utilized. An interesting generalization which gives a useful increase in deductive power with only a small increase in computation is passing sets of possible values around the model rather than single values. This makes it possible to simultaneously reason about alternate possibilities. Each transmission step results in the intersection of the sets of possibilities at the two ends of a link. The constraints on a node are then used to further narrow the possibilities associated with the ports of a node. This in turn, triggers additional set intersections and so on. Eventually, many sets can be reduced to a single element (representing a definite value), even though every set had many elements to start with. In addition, some sets may be reduced to having no elements, which indicates the existence of a contradiction.

The prototypical example of the propagation of possibility sets is Waltz's line drawing interpretation program (see [12]). Given a perspective drawing of a group of blocks and the shadows they cast, he showed that the propagation of possibility sets describing the characteristics of individual lines could be used to identify which subgroups of lines correspond to single objects.

A further increase in power can be obtained by passing symbolic information around in a constraint model. In Example 3 below, symbolic expressions are passed between nodes and simple expression manipulation routines are used to deduce expressions from expressions. This kind of approach can lead to significantly more powerful conclusions, however, it risks getting bogged down in excessively complex local deductions. As a result, it is important to keep the deductive apparatus simple. At the current time, picking the right level of deduction is an art rather than a science, however, there are a number of examples (e.g., [11]) which show that a profitable middle ground exists.

Example 1. Suppose that it is asserted that $V_y = 4$. This triggers a chain of trivial deductions culminating in the conclusion that $V_x < 2$ as follows. First, $V_y = 4 \rightarrow V_j^c = 4 \rightarrow V_j^a = 4 \rightarrow V_r^b = 4$. This in turn implies that $I_r^a = (5-4)/1000 = .001$. (This is consistent with the requirement that $I_r^a < .01$.) Next, $I_r^a = .001 \rightarrow I_j^a = .001$. Since $I_j^c = 0$, $I_j^a = .001 \rightarrow I_j^b = .001 \rightarrow I_t^c = .001$. (This is consistent with the requirement that $I_t^c < .005$.) Finally, $V_y = 4 \rightarrow V_j^c = 4 \rightarrow V_j^b = 4 \rightarrow V_t^c = 4$. (This is consistent with the fact that $V_t^c - V_t^e = 4 - 0 = I_t^c * 4000 = .001 * 4000$.) As a result, the last constraint associated with t implies that $V_t^b - 0 < 2$ which implies that $V_x < 2$.

Example 2. Suppose that it is asserted that $V_y = 3$. In this situation propagation proceeds in exactly the same way as in Example 1. It is concluded that $V_r^b = 3$ and therefore $I_r^a = (5-3)/1000 = .002$. It is then concluded that $I_t^c = .002$. In addition, it is concluded that $V_t^c = 3$. However, this is not consistent with either of the possible states of t, because $.002 * 250 \neq 3 - 0 \neq .002 * 4000$. A contradiction has been found. Under the assumptions of the model, it is not possible for V_y to be equal to 3.

Example 3. Suppose that it is asserted that $V_x = 3$. This triggers a chain of deductions culminating in the conclusion that $V_y = 1$. To start with, $V_x = 3 \rightarrow V_t^b = 3 \rightarrow t_r = 250$. However, no more numerical deductions can be performed. In order to gain any further information from the model, more powerful kinds of deduction have to be applied. As mentioned above, a standard way to do this is to introduce symbolic computation based on the propagation of simple expressions in addition to numerical values. This is significantly more complex than the propagation of numerical values, but still much simpler than general purpose deduction.

In this example, the symbolic value V_y can be propagated through the node j to conclude that $V_y = V_r^b = V_t^c$. Similarly, $I_j^c = 0 \rightarrow I_r^a = I_t^c$. Combined with the other facts which are known, this implies that:

$$\begin{aligned} V_y - 0 &= I_t^c * 250 \\ 5 - V_y &= I_t^c * 1000 \end{aligned}$$

Solving these two equations simultaneously reveals that $V_y = 1$ and $I_t^c = .004$. (In conjunction with the examples above, this example illustrates the bidirectional nature of constraint propagation—information propagates outward from wherever it is to wherever it is not.)

Example 4. Suppose that the resistance of r is 250 ohms instead of 1000 and that it is asserted that $V_x = 3$. Everything proceeds in exactly the same way as in Example 3 until it is determined that:

$$\begin{aligned} V_y - 0 &= I_t^c * 250 \\ 5 - V_y &= I_t^c * 250 \end{aligned}$$

Solving these two equations simultaneously reveals that $V_y = 2.5$ and $I_t^c = .01$. This, however, contradicts the requirement that $I_t^c < .005$. This contradiction has a somewhat different character than the one in Example 2. It reveals a failure in the design of the circuit (the resistance of r must be larger than the value assumed, if t is to be protected from being overloaded), rather than a faulty assumption about the value of a voltage.

Partial modeling. A key thing to notice about the examples above is the importance of the fact that the constraint network in Figure 2 is only a *partial* model of the circuit in Figure 1. The model is partial, because it ignores many aspects of the circuit. It ignores the fact that the wires themselves have resistance. It ignores AC characteristics (i.e., inductance and capacitance). It ignores the fact that there is some current through the base of a transistor. It ignores the fact that the relationship between the voltage drop from the base to the emitter and the resistance from the collector to the emitter is much more complicated than a step function. The simplifications are reflected both in the topology of the model (e.g., the choice of what parts of the circuit become nodes) and in the constraints associated with the nodes.

However, even ignoring all these aspects of the circuit, the model captures the basic non-linearity of the transistor and the fundamental operation of the circuit. As a result, it is possible to draw useful conclusions from the model. There are many conclusions which cannot be drawn, but the ones which can be drawn are basically correct. (In the domain of electronic circuits a considerable body of knowledge has evolved which indicates what kinds of simplifications can safely be used in which situations. In the software examples in the next section, the safety of the simplifications chosen is justified in greater detail.)

The advantage of the simplifications is that they make reasoning about the circuit practical. In principle, it would be possible to draw the same conclusions based on a complete theory of the physics of each device and wire in the circuit. However, performing the required deductions and algebraic manipulations would be enormously complicated.

System Validation via Constraint Modeling

In order to apply constraint modeling to the task of system validation one need merely create a constraint model which describes some aspect of the system and its specification. Constraint propagation can then be used to discover contradictions in the model. Each contradiction discovered corresponds to a failure or error in the system. (Contradictions correspond to errors as well as failures, because constraint models typically contain significant amounts of information about internal behavior as well as external behavior.)

The primary weakness of constraint modeling as a validation tool is that it can only draw conclusions about the things which are in the model. As a result, it can only provide partial validation.

The main attraction of constraint modeling as a validation technique is that its primarily weakness is very different from the primary weaknesses of testing and code inspection. As a result, it is beneficial to combine the three techniques.

Whereas testing has trouble finding failures in unusual situations, constraint propagation investigates every possibility equally without any bias toward usual situations. In addition, while testing is oriented primarily toward checking for failures, constraint propagation checks for internal errors as well, thereby increasing the range of fault symptoms which can be detected.

Whereas code inspection is inherently local, constraint propagation investigates the interaction of information collected from arbitrarily distant points in the model. In addition, while the informal nature of code inspection can cause details to be overlooked, constraint propagation always follows up every detail.

A weakness constraint modeling shares with code inspection is that it is limited in the complexity of the deductions it can make. In particular, constraint propagation is typically not capable of deriving every possible conclusion from the information in the constraints. As a result, constraint propagation can miss a contradiction if it is sufficiently abstruse.

Another interesting aspect of constraint modeling is that, unlike many formal techniques, it is at its best when applied in the large. When applied to small problems, the only real benefit of constraint modeling is that it is thorough and automatic, because in small problems, the relatively simple deductions which are possible are not particularly impressive in comparison with code inspection. However, when applied to large systems, the leverage provided by constraint modeling multiplies as constraint propagation triggers the interaction of information which is widely separated in the model. Further, although constraint propagation can be time consuming, any opportunity to trade off human time for computer time is always worth pursuing.

Current Examples of System Validation via Constraint Modeling

There are two common methods of system validation (type checking and automatic system construction) which can be viewed as constraint modeling. Each of these methods is a good example of how constraint modeling can be applied to programs.

Type checking. Given a subroutine, type checking implicitly constructs a constraint model where each function call, procedure call, and operation is a node. These nodes have ports corresponding to the inputs and outputs of the function, procedure, or operation. The ports are connected by links which reflect the data flow in the subroutine. The model also contains a node for each variable in the subroutine. These nodes have a single port which is connected to every place where the variable is read or written.

In basic type checking, the constraints associated with the nodes in the model all have a particularly simple form. Each constraint specifies a type which is mandatorily linked with a port. For example, an instance of the Fortran function `SQRT` would have constraints that specified that its input and output are both real numbers. A variable `J` might be associated with a constraint which specifies that it is an integer.

The process of checking that the types used in a subroutine are consistent proceeds using constraint propagation as described in the last section. However, this processing is simplified because the constraints have such a simple form. Since each constraint specifies a constant type, every transfer function associated with a node is a constant function. That is to say, the type at a port is always the same no matter what is known about other ports. Further, if the type is known at a port it is always known no matter how little is known about other ports. As a result, information never needs to propagate through nodes. The only thing which is necessary is to check that the type at the end of a link connecting two ports agrees with the type at the other end. The sole result of basic type checking is a list of the places where type conflicts occur. These conflicts can be reported to the programmer or, in some situations, resolved by automatically applying coercions (e.g., when an integer is assigned to a real valued variable).

Even this basic form of type checking is an excellent example of the power of constraint modeling. Very large simplifications are being made. (From the point of view of the model, the functions `SQRT`, `SIN`, and `ATAN` are all identical.) Given the magnitude of this simplification, one would not dream of suggesting that a subroutine that does not contain any type errors must therefore be correct. However, the simplification is conservative in that it is next to impossible for a program which contains type conflicts to be correct. Further, a wide variety of faults have the collateral effect of causing type errors. As a result, a wide variety of faults can be detected using type checking, even though the type error itself is almost never the essence of the fault.

More complex forms of type checking are based on more complicated constraint propagation. Consider the overloading of names. Many programming languages have the feature that a single name can be used for many different things (e.g., several operators or several functions) as long as the individual things can be differentiated based on argument and/or result types. As a trivial example, consider Fortran. The operator "+" stands for a family of operations which add integers, reals, complex numbers, and so on. Which specific operation is meant is determined by looking at the types of the arguments of the operation. This can be looked at as an example of constraint propagation using non-constant transfer functions. The input ports of a "+" node can have several different types. A constraint specifies what the type of the output port is based on the input types. In addition, given the input types, the constraint indicates which specific "+" operation is required.

Probably the most complex support for overloading is provided by Ada [13]. In Ada, function names, procedure names, operator names, and literal data items (e.g., enumeration elements) can all be overloaded. As a result, it is not always possible to resolve overloading by looking at input types. It is sometimes necessary to reason in the other direction by looking at the output type. Thus, bidirectional constraint propagation is used to resolve overloading.

A related example is provided by the language ML [7]. In ML, programmers are not required to specify the types of variables. Bidirectional constraint propagation is used to determine the type of a variable based on the way the variable is used.

Restricting what can be written The discussion above is equally applicable whether or not a language requires strong typing. If strong typing is required, then every port must have a type constraint. If weak typing is allowed, then some ports will not have a user specified type. However, if a large number of ports are untyped, then the type checking model will contain very little redundancy and constraint propagation will not be able to do very much useful checking.

This brings up an important issue. The usefulness of a partial validation approach such as type checking can be greatly enhanced if programmers are willing to cooperate. In most modern languages it has been decided that the programmer's cooperation with type checking should be made mandatory (i.e., via strong typing). In a similar manner, code inspections are facilitated if programmers write code which is clear and concise. (Unfortunately, although coding standards can be helpful, there is no obvious way to make this mandatory.)

Automatic system construction. The prototypical problem attacked by automatic system construction tools is the construction of an up-to-date and consistent, executable version of a system. This is done by using constraint propagation applied to a model which is very different from the one used for type checking (see [2, 9]). In a system construction model there are essentially two kinds of nodes: data objects and processes. The data object nodes

represent units of the system from the standpoint of construction (e.g., a file of functions, a binary file which is the result of compilation, or a grammar file which is used as the input to a parser generator). The process nodes represent operations on these data objects (e.g., compiling, linking, or generating a parser). Each process node has a number of input and output ports and is annotated with a statement which specifies how to actually perform the operation. The links between the nodes specify which data object is created by each process and which processes use each data object.

The constraints on the nodes have an extremely simple form. Each object node is annotated with the time at which it was last modified. Each process node is given a constraint that requires the time associated with the output object to be greater than the maximum time associated with any input object.

In order to construct a consistent version of the system, constraint propagation is applied in order to determine where the constraints on the process nodes are violated. Whenever one of these constraints is violated, a record is made of the fact that the process must be run, and the time associated with the output object is increased to reflect the fact that it will be recreated. If any process nodes use this object, then this change of time will lead to additional constraint violations. The output of the propagation step is a list of processes which have to be run in order to create a consistent version of the system.

The determination of what processes must be run can be done very efficiently, because the constraint on the process nodes is inherently unidirectional and the construction model must be acyclic. As a result, a single sweep over the model suffices to determine all constraint violations.

As with type checking, it is interesting to note that constraint modeling yields very useful results even thought the model contains extremely little information about the system and the way it is modified. (From the point of view the model, every different kind of modification is the same. The only relevant feature is the time at which it is performed.) Given the magnitude of this simplification, it is not surprising that automatic system construction tools can only approximate the goal of running the construction processes when and only when necessary. However, the simplification is carefully chosen to be conservative in that one can be confident of the fact that a processes never needs to be run when its constraint is satisfied.

As has been suggested by a number of researchers (see [9]), an obvious direction in which to extend automatic system construction tools would be toward having a more detailed constraint model. In particular, the kinds of modifications could be modeled in more detail. For example, a record could be kept of all the changes to each data object. This record would show the nature of each modification and the time it took place. More complex constraints could then be used to determine when processes need to be run. For example, a compilation does not have to be performed if all the changes to a source file are confined to comments.

More General Uses of Constraint Modeling

Both type checking and automatic system construction are very simple examples of constraint propagation. System validation could be enhanced by using more complex constraint propagation.

Extending type checking. There are many ways in which type checking can and is being extended. For example, it is very useful to apply type checking at the inter-module

level based on interface specifications as well as at the intra-module level. The constraint modeling perspective suggests two additional directions of extension.

Consider basic type checking. From the point of view of constraint modeling, the essence of type checking is the fact that the transfer functions are constant functions. The efficiency of type checking algorithms stems directly from this fact rather than anything to do with types *per se*. The same algorithms could be used to check for the consistency of anything which can be expressed in terms of constant transfer functions. There is no need for the analysis to be restricted to what is commonly thought of as a *type*. One should think, "what is the most specific assertion which can be made", rather then merely thinking, "what is the type of this object".

Abandoning the usual connotation of the word type suggests a second extension. The term *type* has become associated with the idea that every object should have a single type. However, there is no need for things to be so limited. Rather, each object could be tagged with several type-like features. Type checking could be simultaneously applied to each of these features. For example, each object might be given a representational type essentially similar to what is currently considered to be a type and additional type features indicating how the object should be used (e.g., information about modifiability or security issues).

Although only modestly more powerful than what is done now, the extensions suggested above represent a more general point of view which could be useful. Both of the extensions could easily be generalized to cover the resolution of overloading and type determination as in ML.

Dimensional analysis. A particularly interesting example of how one could move beyond simple notions of type is the often suggested (but apparently seldom implemented) idea of extending type checking to including dimensional analysis [4, 5, 6, 8]. Given the usefulness of dimensional analysis when working with equations in physics and engineering, dimensional analysis aught to be quite helpful when validating mathematical software. The essence of dimensional analysis is the observation that, in order for an equation to make physical sense, the two sides of the equation must have the same dimensionality. For example, if one side of an equation is the product of two lengths, then the other side of the equation must also have the net dimensionality of an area. In a programming context, this example could be rephrased by saying that the product of two lengths can only be assigned to a variable which is intended to hold an area.

It has been suggested that dimensional analysis could be supported by extending numeric types so that they contain dimensional information. However, this would be quite cumbersome since it would lead to a proliferation of types representing the cross product between the basic numeric types and the possible dimensionalities (e.g., integer areas, real areas, double precision areas, etc.). It would be better to introduce dimensionality as a separate type-like feature. The storage types could be handled in the standard way while the dimensional types were handled using slightly more complex constraint propagation.

In order to support dimensional analysis, the nodes corresponding to variables in the type checking model would be annotated with constraints stating the dimensions of the value stored in the variable. (If strong dimensional typing was desired, then every variable declaration would include a statement of dimensionality.)

Each computational node in the model would be annotated with constraints specifying the effects of the node on dimensionality. For example, the operation "*" would have a constraint stating that the dimensions of the output are the product of the dimensions of

the inputs (e.g., the product of two areas has the dimensions of length to the fourth power). As in the example in Figure 2, this constraint leads to multiple transfer functions. For example, if the output is an area and one input is a volume, then the other input must have the dimensions of one divided by length.

The inputs and outputs of user functions might well be required to have specific dimensionalities. However, user functions that are more complex from a dimensional standpoint could be supported as long as some method was provided so that the user could state the dimensions of the result as a function of the dimensions of the inputs.

In order to perform a dimensional analysis to check the validity of a program, one would only need to apply constraint propagation to the dimensional information. The dimensional information associated with variables would be propagated through operations and compared with the dimensional information attached to other variables and to user functions requiring specific dimensionalities.

(The effect of a dimensional analysis could be obtained using standard type checking by treating an operation such as "*" as an overloaded operation with one specific instance corresponding to every dimensionally different pair of inputs. However, there are too many different possibilities for this to be a practical approach. It would be much easier to simply use a more powerful notion of constraint propagation.)

Reasoning about units. A third kind of type-like information which is related to dimensions are units (e.g., feet, inches, seconds). Information about units can be propagated in essentially the same way as information about dimensions. However, it would be profitable to handle units separately, because while automatic coercion cannot be applied to dimensions, it can be applied to units. For example, the product of two lengths in feet cannot be compared with a product of a length in feet and a time in seconds, however, it can be compared with the product of two lengths in inches if an appropriate conversion is done.

Estimating numerical error bounds. Another interesting application of constraint propagation which would be somewhat similar to dimensional analysis would be the estimation of numerical error bounds. It is straightforward to state for each operation what the error bounds of the result is, given the error bounds (and representational precision) of the inputs. It would be possible to declare similar information in conjunction with user defined functions. Given this information, it would be straightforward to determine a set of error bounds for every quantity computed in a subroutine as long as the number of iterations of each loop were independent of the input data. This could be used to compute worst case estimates of the error bounds to be associated with user functions.

The computed error bounds could be compared with declared minimal requirements, or simply computed in the style of ML. They could also be used to check that the program does not print meaningless digits, or run the risk of computing a value which does not have any meaningful digits.

Estimating time bounds. Similarly, if the number of iterations of the loops in a program were independent of the input data, a constraint modeling approach could be used to estimate the amount of time required for programs to execute based on estimations of the time required by primitive operations. This approach might be particularly helpful when analyzing real time systems. In particular, it could be used to estimate whether or not a system could handle its intended throughput.

Domain Specific Models

The examples of constraint modeling above have the virtue that they are all domain independent. That is to say, they can be applied to any kind of program as well as any other kind. This gives them wide applicability. However, they are all relatively shallow in that they cannot easily take advantage of domain knowledge. It should be possible to obtain more comprehensive validation results by building a model which is tailored to a particular kind of system and contains significant amounts of domain knowledge.

Given a large system of programs, one would look at the specifications for the system and construct a constraint model which encodes a small but significant portion of the specifications in the form of constraints which are amenable to propagation. It is expected that this model would be closely adapted to the characteristics of the particular system rather than following some rigid pattern.

Under the assumption that type checking and dimensional analysis encode perhaps 1% of the specifications for a system. The basic goal is to use every degree of freedom possible in order to encode perhaps 5% of the specifications in a more complex (but still tractable) set of constraints.

Research is currently under way in an attempt to show the efficacy of this approach. Given that this research is not yet complete, it is difficult to lay out the approach in full detail. However, the basic features of the approach can be summarized.

Designing a type of constraint model for a class of systems. When designing a constraint model, there are two fundamental degrees of freedom: the form of the model and the kind of information which is to be expressed as constraints. Given these decisions, the exact model and constraints follow from the system being modeled.

When considering a system of subroutines, the most obvious kind of model to use is one which is based on the data flow in the system (i.e., a model similar to the one used by type checking). However, this is not the only possibility. It might be better to organize the model around gross data flow between modules or around control flow. In any event, there must be some simple uniform criterion for determining what units in the system (e.g., function calls, subroutines, subsystems) should become nodes in the model and what connections between these units should become links in the model.

The most fundamental decision when designing a constraint model is the choice of what kind of information to encoded as constraints. In particular, the form of the constraints determines the character of the units in the system which will become nodes in the model. In addition, the links in the model follow from the way in which the constraints are intended to propagate.

As illustrated by the examples above, there are many kinds of information (e.g., type information and dimensionality) which should be generally useful when modeling almost any kind of system. However, it is expected that the greatest leverage will come from picking kinds of information which are intimately associated with the specific system to be modeled. By the very nature of this information, it is impossible to describe it in general. However, several criteria should be kept in mind.

To start with, the information must be relevant to a relatively wide range of units in the system. This is true, because the great strength of constraint propagation is its ability to rub together information from a wide variety of locations in the model. In order for this to work to maximum effect, each class of information must be present at a wide variety of locations

in the model. As an extreme example, suppose that a certain class of information is present at only one node in a model (e.g., only one node says anything about dimensionality). In this situation, there is nothing which constraint propagation can do with this information except copy it to other nodes.

Equally important, the information must be amenable to propagation. Although this is certainly of paramount importance, it is not clear to what extent there are any quantifiable criteria for determining what kinds of information are easily propagatable. It is hoped that the current research will yield some insights in this area.

Although still somewhat nebulous, there is one characteristic common to several of the examples in this paper which seems to facilitate propagation. It is apparently beneficial to consider quantities which take on a relatively small number of discrete values rather than a large number of possible values. If a large number of values are possible, then it is seldom practical to reason about specific values. One either has to reason symbolically about possible values (which often introduces complexities which are beyond the current capabilities of automatic deduction) or resort to reasoning qualitatively about classes of values (which reduces the problem to one of reasoning about quantities which take on a small number of values.)

Building a specific constraint model for a particular system. The creative part of constraint modeling revolves around the choice of the kind of information to represent and the way to represent it. Once this has been done, the actual construction of a model should be more or less straightforward.

Given the form of the model, it should typically be possible to derive the detailed pattern of nodes and links more or less automatically from the code for the system being modeled. (This is the case in all of the examples in this paper.) Deriving the constraints associated with the nodes is a more difficult task.

In the absence of a machine readable specification, the constraint derivation process is of necessity a manual one, because the information in the constraints comes primarily from the specification for a system rather then from its code. For example, in order to use type checking, the programmer must explicitly declare a significant amount of type information.

Although the constraint derivation process is expected to be manual, it need not be overly difficult. Given the classes of information which have been chosen, one need only look at each unit of the system which corresponds to a node in the model and develop constraints for the unit based on its specification.

The process of deriving a set of constraints for a unit can naturally be divided into two phases. The first phase is the determination of what the constraints should be. This is almost certainly not automatable and is intimately tied up with the process of developing the specification for the system. For each class of information which has been selected for modeling, the relevant set of constraints essentially encode the answer to the question "how does the unit effect this class of information?".

The second phase is verifying that the constraints correctly describe the unit in question. In the near term it is expected that this task will also be manual. It would be natural to include it as part of the code inspection process. However, it is quite possible that this part of the process could eventually be automated. Since it involves proving relatively small theorems about small programs it may be within the reach of automatic theorem proving techniques relatively soon. At any rate, it is much more realistic as a potential application of these techniques than full-scale verification.

A key feature of both phases of the constraint derivation process is that they are inherently local in nature and therefore in tune with human information processing capabilities. An important way to look at the benefits of constraint propagation is that it bootstraps locally validatable information into global conclusions.

Partial verification. It is interesting to consider the relationship between constraint modeling and verification. If, taken in their entirety, the constraints in a model are complete with regard to some class of information, then the failure of constraint propagation to find any contradictions can often be taken as formal proof of correctness with regard to this class of information. For example, if strong typing is applied to a program and no type conflicts exist, then it can be guaranteed that although there may well be errors in the program, these errors do not involve inappropriate operations on data.

Robustness in the face of incompleteness. Although completeness is clearly beneficial, it is difficult to achieve. An important benefit of constraint modeling is that it is very tolerant of incompleteness. If a constraint is left out, nothing catastrophic happens. The only effect is that some deductions will no longer be possible. This degrades the ability of constraint propagation to find failures, but in a gradual way.

There are many reasons why it is next to impossible to achieve completeness. Most fundamentally, specifications tend to be incomplete. Since the constraints are derived from the specification, they tend to be incomplete as well. Further, there are plenty of opportunities for leaving out one or more constraints for essentially inconsequential reasons.

In addition, it may be necessary to intentionally leave out some constraints because they are too complex to usefully express. The fact that constraint modeling can tolerate this provides an important degree of freedom when devising a propagatable representation for a class of information. It is permissible (and useful) to devise a representation which only works 90% of the time. (This probably degrades the performance of constraint propagation by 50% or more, but still allows constraint modeling to be beneficial.)

A Research Plan

To date, the research presented here has been confined to thought experiments. At the very least, this investigation has revealed that constraint modeling is a useful basis for understanding some of the validation methods currently in use. Going beyond this, it has suggested a provocative direction in which to look for new validation methods.

The next step is to test these ideas by applying them in real world situations. A large real time control system has been obtained and efforts are underway to create a constraint model for it which is useful for validation. Since this system has been very thoroughly tested, there may not be any faults in it that can be easily detected by any means. Experiments will be performed by introducing new faults into the system which are difficult to detect by testing and code inspection, and seeing which of these faults are easy to detect using constraint propagation. If successful, these experiments will show that domain specific constraint modeling can be a useful validation tool.

If these experiments are successful, the next step in the research will be to analyze the experiments with particular attention to determining what kinds of information can profitably be represented as constraints and which cannot. (Depending on the amount of symbolic computation which is required, implementing a system which does the actual propagation is relatively easy (see [10]).) The long term goal will be the creation of a

comprehensive theory of how to distill a significant part of a specification for a system into propagatable constraints. However, it should be noted that domain specific constraint modeling could be a very important validation method even if it has to be applied on a case by case basis rather than guided by a comprehensive theory.

References

[1] A. Avižienis & J. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", *IEEE Computer*, 17(8):67–80, August 1984.

[2] E. Borison, "A Model of Software Manufacturing", in *Advanced Programming Environments, Proceedings of an International Workshop*, Conradi R., Didriksen M., and Wanvik D.H. (editors), *Lecture notes in Computer Science*, 244:187–215, Springer-Verlag, June 1986.

[3] A. Borning, "The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[4] A. Dreiheller, M. Moerschbacher, & B. Mohr, "PHYSICAL: Programming Pascal with Physical Units", *ACM Sigplan Notices*, 21(12):114-123, December 1986.

[5] N.H. Gehani, "Units of Measure as a Data Attribute", Computer Languages, 2:93-111, 1977.

[6] N.H. Gehani, "Ada's Derived Types and Units of Measure", Software—Practice and Experience, 15(6):555-569, June 1985.

[7] M. Gordon, R. Milner, & C. Wadsworth, *Edinburgh LCF: A Mechanical Logic of Computing*, Lecture notes in Computer Science, 78, Springer-Verlag, 1979.

[8] R. Manner, "Strong Typing and Physical Units", *ACM Sigplan Notices*, 21(3):11-20, March 1986.

[9] R.E. Robbins, *Build: A Tool for Maintaining Consistency in Modular Systems*, MIT MS Thesis, MIT/AI/TR-874, November 1985.

[10] G. Steele, *The Definition and Implementation of a Computer Programming Language based on Constraints*, MIT PhD Thesis, MIT/AI/TR-595, August 1980.

[11] G.J. Sussman & R.M. Stallman, "Heuristic Techniques in Computer-Aided Circuit Analysis", *IEEE Transactions on Circuits and Systems*, 22(11):857–865, November 1975.

[12] P.H. Winston, *Artificial Intelligence*, second edition, Addison-Wesley, 1985. (See Chapter 3.)

[13] *Military Standard Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, February 1983.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 5/5/95

Report # AIM-1020

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 19(24-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UN#'ED TITLE PAGE	
(2-19) PAGES #'ED 1-18	
(20-21) SCANCONTROL, DOD	
(22-24) TARGETS (3)	

Scanning Agent Signoff:
Date Received: 5/5/95 Date Scanned: 5/9/95 Date Returned: 5/11/95

Scanning Agent Signature: Michael W. Cook

REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1988	

4. TITLE AND SUBTITLE
System Validation via Constraint Modeling

5. FUNDING NUMBERS
NSF IRI-8616644,
IBM, NYNEX,
Siemens, ONR
N00014-85-K-0124

6. AUTHOR(S)
Richard C. Waters

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
AIM 1020

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Information Systems
Arlington, Virginia 22217

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AD-A193589

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Constraint modeling could be a very important system validation method, because its abilities are complementary to both testing and code inspection. In particular, even though the ability of constraint modeling to find errors is limited by the simplifications which are introduced when making a constraint model, constraint modeling can locate important classes of errors which are caused by non-local faults (i.e., are hard to find with code inspection) and manifest themselves as failures only in unusual situations (i.e., are hard to find with testing).

14. SUBJECT TERMS
constraints, validation, modeling, testing

15. NUMBER OF PAGES
19

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**